Automated software protection for the masses against side-channel attacks

NICOLAS BELLEVILLE, Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
DAMIEN COUROUSSÉ, Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
KARINE HEYDEMANN, Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
HENRI-PIERRE CHARLES, Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France

We present an approach and a tool to answer the need for effective, generic and easily applicable protections against side-channel attacks. The protection mechanism is based on code polymorphism, so that the observable behaviour of the protected component is variable and unpredictable to the attacker. Our approach combines lightweight specialized runtime code generation with the optimization capabilities of static compilation. It is extensively configurable. Experimental results show that programs secured by our approach present strong security levels and meet the performance requirements of constrained systems.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; **Software security engineering**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Side-channel attack, hiding, polymorphism, compilation, runtime code generation

ACM Reference format:
Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. 2017. Automated software protection for the masses against side-channel attacks. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (January 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Physical attacks represent a major threat for embedded systems; they provide an effective way to recover secret data and to bypass security protections, even for an attacker with a limited budget [31, 32]. Side-channel attacks consist in observing physical quantities (power, electromagnetic emissions, execution time, etc.) while the system is performing an operation [26]. They can easily recover a secret after a few hundreds of observations or less. In the literature, side-channel attacks are most of the time associated to cryptography because they represent the most effective threat against implementations of cryptography. In the previous decades, side-channel attacks targeted specific products such as Smart Cards, and countermeasures were applied manually by dedicated security experts. Today security components are embedded everywhere, and as a consequence side-channel attacks become a threat for many of everyday life objects, for example light bulbs [32] or bootloaders [37]. Thus, there is a strong need for automated solutions able to tackle everywhere the issue of side-channel attacks.

Many software and hardware countermeasures against side-channel attacks have been developed, but most of them are *ad hoc* and require specific engineering developments considering either the

DOI: 10.1145/nnnnnnn.nnnnnnn

hardware or the application code to protect (for example masking). Several works have shown that polymorphism is an effective software countermeasure against side-channel attack [5, 8, 18, 19]. The idea is to obtain a different behaviour from one execution to the next one so that each side-channel observation differs, thus effectively increasing the difficulty to recover the secret data. Tools have been proposed by researchers to help developers to apply polymorphism on their code. Polymorphism was implemented by generating code variants statically (*multi-versioning*) [7], or by generating code variants at runtime [5, 18]. When variants are generated statically, the number of variants is limited by the final size of the program, as generating more variants induces an increase of code size. By contrast, tools that use runtime generation suffer from other drawbacks: (1) runtime code generation is usually avoided in embedded systems because of the potential vulnerability introduced by the need to access some segments of program memory with both write and execution permissions; (2) lightweight runtime code generation lacks genericity, e.g., is applied on JIT-generated code only [24] or relies on a Domain Specific Language [18].

In this paper, we present a generic approach supported by a tool, named Odo, which enables to automatically protect any software component against side-channel attacks with runtime code polymorphism. Our key idea is to base the polymorphic code generation on specialized runtime generators, which can only generate code for the targeted function to harden. Furthermore, our approach leverages compilation to automatically generate the specialized generator for any function specified by the developer. Specialisation with compilation reduces the computational overhead incurred by runtime code generation. It takes advantage of a compilation flow to gather static information and to optimize the code produced at runtime. Specialisation with compilation also enables a precise static allocation of memory. As a consequence it makes possible the deployment of mitigations to the concerns related to runtime code generation in embedded systems (i.e., restrict write permissions on program memory), and the use of the proposed approach in embedded systems with limited memory resources.

At runtime, the specialized generators use the available static information and several runtime code transformations to generate a different code efficiently and periodically. Some transformations have already been shown effective against several types of side-channel attacks: register shuffling, instruction shuffling, semantic variants and insertion of noise instructions. The specialized generators can also use a new and so-called dynamic noise instructions to introduce more variability even between consecutive executions of the same generated code. As every transformation can be enabled/disabled or tuned, the proposed approach offers a high level of polymorphism configurability.

In the experimental results, we first analyse in details an AES use case by considering 17 different configurations of polymorphism among the large set of possible configurations owing to the configurability of our approach. We assess the security level of the hardened AES with two different evaluation criteria nowadays in use for the evaluation of side-channel countermeasures: non-specific t-tests, to assess the absence of information leakage, and Correlation Power Analysis (CPA). The security evaluation based on t-tests show that several levels of security can be reached. We also analyse the impact of the different transformations on security and performance. This gives some insight on ways to satisfy some security and performance requirements. Finally, we present a methodology to find a configuration leading to a good trade-off between security and performance. Following it, we select 3 configurations with different security and performance trade-offs. The results of a CPA attack that targets the weakest of these configurations in terms of security revealed that attacking it is 13,000 folds as hard as attacking the reference unprotected implementation. As our approach is fully automatic, we also evaluate the code size and runtime overheads considering 15 benchmarks and the 3 selected configurations. The evaluation shows that (1) the overheads are small enough so that our approach is applicable even on highly constrained systems and (2) it is very

competitive compared to the state of the art. Indeed, code generation is highly efficient and is an order of magnitude faster than similar state-of-the-art approach.

Thus, experimental results demonstrate the versatility and the strength of our approach: it matches the needs in terms of security, thanks to a high behavioural variability, while incurring an acceptable performance overhead; its high configurability enables to adjust performance and security levels for a particular case, such that polymorphism can be deployed easily on a wide variety of programs; it removes the traditional concerns about runtime code generation, reaching the same confidence level as static multi-versionning approaches with lower overheads.

This rest of this paper is organized as follows: Section 2 gives some background on side-channel attacks and existing software protections, and Section 3 details our threat model. Our approach and its implementation in Odo are presented in Section 4; Section 5 is dedicated to the memory management. The experimental evaluation is presented in Section 6. Section 7 is devoted to a comparison with the closest existing approaches. Related work are presented in Section 8 before concluding in Section 9.

2 BACKGROUND

Cryptographic components that are currently in use in almost every computing system, such as AES or RSA, are considered to be secure from the point of view of cryptanalysis, but Kocher *et al.* [26] showed that an attacker can recover a secret cipher key by analysing the power consumption of a device. Such attacks, named side-channel attacks, exploit the fact that the physical emissions of the device are dependent on the executed instructions and the processed data. In order to extract exploitable information from the measurements, the attacker computes a model of the energy consumption of the device for every possible value of the secret she wants to disclose. Then, she compares the measurements with the modelled values, typically using a correlation operator. The modelled value that matches best the measurements then leads to the secret key value. The side-channel analysis takes a divide-and-conquer approach so that the attack is computationally tractable: usually each data byte is recovered separately. In our experimental validation, we used near-field electromagnetic measurements, but the principle of the attack is similar for EM radiation and power consumption measurements.

The two main protection principles against these side-channel attacks are masking and hiding.

Masking consists in combining the sensitive (key dependent) intermediate computation data with random values so that side-channel observations are unpredictable to the attacker. Hiding consists in blurring the side-channel observations usually by introducing a kind of behavioural randomization, for example in the amplitude of the measurements or with timing desynchronisations, in order to increase the difficulty for an attacker to find exploitable information. Masking is an algorithmic protection; it is currently the subject of a lot of research in cryptography because there is no general solution to this protection principle; as a consequence masking schemes have to be designed carefully for every sensitive algorithm. In practice, to the best of our knowledge, masking is still manually applied on industry-grade secured components, which is error prone in addition to be costly. Plus, higher-order attacks are effective against masked implementations [26]. Going one step further, Moos & al showed that higher-order leakages can be turned into first-order leakages with a simple filtering, enabling the use of first-order attack on masked implementations [29]. In practice, masking is combined with one or several hiding protections in industry-grade products.

Hiding countermeasures are various, like random delays [16, 17], random dynamic voltage and frequency scaling [9, 36, 38], dynamic hardware modification [33], or code polymorphism [18], which is the countermeasure used in this paper. Contrary to masking that removes information leakage up to a d-order, polymorphism do not remove information leakage from side-channel observations. Yet, it was demonstrated as an effective solution to decrease the exploitability of information leakage so that

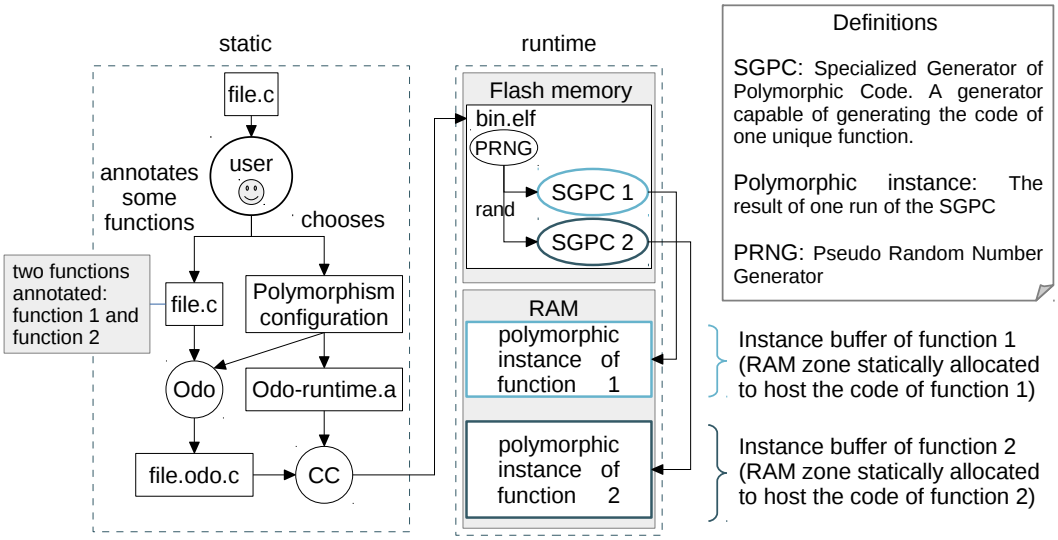

Fig. 1. The overall compilation flow. The role of the user is reduced so that our approach can be deployed as fast as possible on pre-existing code base. The polymorphic functions' binaries are generated at runtime by their respective SGPCs.

side-channel attacks are harder to perform [5, 7]. The work presented in this paper aims to provide an automatic, user-friendly and secure solution to the use of polymorphism in embedded software.

3 THREAT MODEL

In this paper, we target side-channel attacks that exploit either power consumption or electromagnetic emission. We assume that the attacker can control program inputs (e.g. to perform text chosen attacks), and that she has access to the output of the program. Yet, we consider that the attacker cannot get control over the random number generator. This assumption is common for side-channel attacks countermeasures, as masking and hiding rely on the random number generator.

4 AUTOMATIC APPLICATION OF POLYMORPHISM

4.1 Overall flow

In this section, we give an overview of our approach, which is illustrated in Figure 1. The overall approach is divided in two parts: a static part and a runtime one.

The user starts by annotating the target functions to be secured with polymorphism. Then, he chooses a configuration of polymorphism. The choice of such a configuration will be discussed in section 6.2. The annotated C file (*file.c* in Figure 1) is compiled by our tool, Odo, into another C file (*file.odo.c* in Figure 1). The code of each function to secure has been replaced by: (1) a wrapper that interfaces with the rest of the code, and (2) a dedicated generator. The wrapper handles the calls to the dedicated generator and to the produced code. The wrapper is also in charge to make the link between the original function, identically called by the rest of the code, and the polymorphic instance, i.e. to call the polymorphic instance with the right arguments and to return its return value. One dedicated generator is created for each polymorphic function; the implementation of the generator, which is automatically generated by Odo, is entirely dependant on the initial code of the function,

and also depends on the polymorphic configuration chosen by the user. We refer to these generators as *SGPC* (Specialized Generator of Polymorphic Code) later on.

The produced C file is then compiled along with the Odo-runtime library, which provides the architecture support and the code transformation framework, into a binary file that is then loaded on the platform. A dedicated zone in RAM is statically reserved to host the runtime-generated code of a polymorphic function. This memory zone is called an *instance buffer*. In Figure 1, two functions are annotated in the source code. As generators are specialized, there are one SGPC and one instance buffer for each annotated function.

At runtime, the SGPC is called by the wrapper whenever the code has to be regenerated in order to generate a new *polymorphic instance* in the instance buffer. Calling it regularly gives the property of polymorphism to the function: its code changes at each call. The frequence of the regenerations can be controlled by the user. We call *regeneration period*, denoted as ω, the number of consecutive executions of the same polymorphic instance before a new regeneration. When the SGPC is called, the permissions of its instance buffer are switched from execute-only to write-only, and switched back at the end of generation. This ensures that the instance buffer is never writable and executable simultaneously (Section 5.3).

4.2 Generation of Specialized Generators of Polymorphic Code

In this section, we present how Odo generates a SGPC for a function chosen by the user. The runtime code transformations implemented to leverage code polymorphism are presented in the next section. Odo is a standard compiler, based on LLVM. It performs a normal compilation in order to produce a suite of assembly instructions corresponding to the targeted function body, and then it generates the SGPC dedicated to this function from this suite of instructions. The code of a SGPC is composed of a sequence of calls to binary instruction emitters that targets the sequence of ARM assembly instructions generated by the normal compilation flow.

In Odo, the generation of SGPCs is executed by a new backend, which is fully identical to the ARM backend except for the code emission pass. As only this pass differs between our backend and the ARM backend, the suite of instructions from which a SGPC is built benefits from all previous optimisations of the compiler. The emission pass emits the C code of the SGPCs of the annotated functions instead of emitting assembly code. The SGPCs produced with polymorphism activated are quite similar to the ones obtained with polymorphism disabled, the differences are highlighted in the next section.

Listing 3 represents the code generated by Odo for the function annotated in Listing 1, when polymorphism is deactivated. This code is composed of a SGPC for `f_critical` named `SGPC_f_critical` and a new function `f_critical` which interfaces with the rest of the code. The SGPC of `f_critical`, `SGPC_f_critical`, is designed to emit a suite of binary instructions identical to the assembly code that LLVM would have generated for the function (Listing 2). The SGPC is composed of a suite of function calls to the Odo-runtime library, including one call for each binary instruction to be generated. The encoding of all the ARM Thumb1 and Thumb2 instructions are available through the library. For instance, in Figure 3, the call `eor_T2(r[4],r[1],r[0])` writes in the instance buffer the binary instruction `eor r4,r1,r0`. The suffix `_T2` indicates that the Thumb2 encoding is used. All the binary instruction emitters defined in the `Odo-runtime.a` library (Figure 1) take the instruction operands, physical register names and/or constant values, as parameters, as would be for regular machine instructions. This enables the SGPC to change the operands from one generation to another one (e.g., `r[4]` can refer to a different physical register).

In addition, the SGPC raises interruptions at the very beginning and the very end of its execution so that the interrupt handler changes the access permissions of the dedicated instance buffer. This is illustrated by the calls to *raise_interrupt_rm_A_add_B* functions in Listing 3. In practice, these

interrupt calls are inlined using assembly primitives. The mechanisms enabling memory permissions management are presented in details in Section 5.3.

Listing 1. Original C file annotated by the user

```
#pragma odo_polymorphic
int f_critical(int a, int b) {
        int c = a^b;
        a = a+b;
        a = a % c;
        return a;
}
```

Listing 2. ARM instructions suite generated by LLVM for `f_critical`

```
f_critical:                 @r0←a; r1←b
        push r4, pc
        eor r4, r1, r0      @r4←r1^r0
        add r0, r1, r0      @r0←r1+r0
        sdiv r1, r0, r4     @r1←⌊r0/r4⌋
        mls r0, r1, r4, r0  @r0←r0−r1∗r4
        pop r4, lr          @return r0
```

Listing 3. The C file generated by Odo when all polymorphism transformations are turned off

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
        raise_interrupt_rm_X_add_W(code_f);
        reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
        push_T2_callee_saved_registers();
        eor_T2(r[4], r[1], r[0]);
        add_T2(r[0], r[1], r[0]);
        sdiv_T2(r[1], r[0], r[4]);
        mls_T2(r[0], r[1], r[4], r[0]);
        pop_T2_callee_saved_registers();
        raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
        if (SHOULD_BE_REGENERATED())
                SGPC_f_critical();
        return code_f(a, b);
}
```

Listing 4. The C file generated by Odo when all polymorphism transformations are activated

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
        raise_interrupt_rm_X_add_W(code_f);
        reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
        shuffle_regs(r);
        push_T2_callee_saved_registers();
        gennoise(); variant_eor_T2(r[4], r[1], r[0]);
        gennoise(); add_T2(r[0], r[1], r[0]);
        gennoise(); sdiv_T2(r[1], r[0], r[4]);
        gennoise(); mls_T2(r[0], r[1], r[4], r[0]);
        gennoise(); pop_T2_callee_saved_registers();
        raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
        if (SHOULD_BE_REGENERATED())
                SGPC_f_critical();
        return code_f(a, b);
}
```

Listing 5. Example of an ARM assembly code generated by `SGPC_f_critical` with all polymorphism transformations activated.
r5 is used instead of r4 due to the register shuffling, a semantic variant is inserted for the exclusive or and several noise instructions were inserted. No instructions were shuffled due to the dependencies between the instructions.

```
f_critical:
        push r5, r7, r8, r9, lr
        eor r5, r1, #42  @semantic variant used
        eor r5, r5, r0
        eor r5, r5, #42
        add r0, r1, r0
        sub r9, #127     @noise instruction
        sdiv r1, r0, r5
        add r7, r9, #5   @noise instruction
        eor r8, #3       @noise instruction
        mls r0, r1, r5, r0
        pop r5, r7, r8, r9, pc
```

4.3 Runtime code transformations and their generation

In this section, we present the code transformations used to generate a different code each time the SGPC is called. We explain how compilation flow is leveraged to enhance the code transformations at

runtime without requiring costly code analysis. We also present how the code of the SGPC generated by Odo differs when these transformations are activated.

Five different transformations can be used by the SGPCs to vary the code of polymorphic instances: (1) register shuffling, which is a random permutation among the callee-saved registers, (2) instructions shuffling, which consists in emitting in a random order independent instructions, (3) semantic variants, which refers to a random replacement of some instructions by a sequence leading to the same result, (4) noise instructions, which are useless instructions inserted in between the original instructions of the function, and (5) dynamic noise, which consists of a sequence of noise instructions preceded by a random forward jump so that the number of executed instructions varies at each execution.

Listing 4 shows the output of Odo for the input of Listing 1, when all polymorphic transformations are activated. Listing 5 is an example of a polymorphic instance that can be generated by the SGPC at runtime.

Register shuffling. Contrary to what was proposed previously [18] where random register allocation was performed at runtime, here register allocation is done statically by the compiler, resulting in a better allocation and a faster runtime code generation. SGPCs make registers vary by relying on a permutation, which is done at the beginning of each code generation (`shuffle_regs` in Listing 4), between the general purpose callee saved registers (r4-r11), and which is a fast operation. Only instructions that encode registers on 4 bits are used. The effects of register shuffling are illustrated in Listing 5: in this example, register r5 is used instead of register r4.

Instruction shuffling. This transformation aims at shuffling independent instructions prior to their emission in the instance buffer. The emitted binary instructions are first stored in a temporary buffer, named *shuffling buffer*, of configurable size (32 instructions in our experiments). Each binary instruction is associated with its *defs* and *uses* registers. Before an instruction is added to the shuffling buffer, Odo-runtime performs a dataflow analysis, starting from the last instruction in the shuffling buffer, to compute the list of possible insertion locations. A random location is then selected among the possible insertion locations. The shuffling buffer is flushed into the instance buffer at the end of each basic block or when it is full. The instruction shuffling transformation is transparently carried out by Odo-runtime, the code Odo generates for the SGPC is identical whether the transformation is activated or not.

Semantic variants. Some instructions can be replaced by a suite of instructions that achieves the same result and leaves all the originally alive registers unmodified (status registers included). Odo-runtime currently provides semantic variants for instructions that are frequently used in cryptographic ciphers to manipulate sensitive data: instructions belonging to the families of `eor`, `sub`, `load` and `store`, it can be easily extended. Currently, each original instruction can be replaced by 1 to 5 variant instructions. Odo generates specific function calls to the Odo-runtime library for the emission of these instructions when semantic variants is activated. In Listing 1, green bold calls are in charge of the emission of semantic variants. At runtime, the SGPC emits the binary code of one variant randomly chosen among available ones (which include the initial instruction). The call `variant_eor_T2(r[4], r[1], r[0]);` from Listing 4 can generate the original instruction `eor r4, r1, r0` or, e.g., a sequence `eor rX, r0, #rand; eor r4, r1, #rand; eor r4, r4, rX` (as illustrated in Listing 5) where `rX` is a randomly chosen free register and `#rand` is a random constant. Semantic variants for arithmetic instructions (e.g., sub and xor) are based on arithmetic equivalences, variants for stores use smaller stores like store-bytes or store-halfwords, and variants for loads use unaligned loads in addition of load-bytes and load-halfwords.

Noise instructions. We call *noise instructions* functionally-useless instructions that are generated in between the useful instructions. The insertion of noise instructions is performed by the calls to

gennoise (in blue italic in Listing 4). Similar to other transformations, Odo generates such calls only when this code transformation is activated.

The side-channel profile of noise instructions should be as close as possible to the profile of useful instructions, so that the attacker cannot distinguish them and filter them out from the side-channel measurements [21]. We selected noise instructions among instructions that are often used in programs, such as addition, subtraction, exclusive or, and load. The user can specify a particular range of addresses for the loads, which can allow random loads from the AES SBox for instance. Otherwise, a small static random table is used.

Each insertion of noise instructions is guided by a probability model. Odo-runtime currently offers two different configurable models. In both models, a random draw determines if noise instructions are inserted or not. The models are presented in Table 1. The number p is the probability of insertion of one or more noise instructions, and P[X=i] represents the probability of inserting i noise instructions. The parameter N controls the maximum number of noise instructions that can be inserted at once. The first model, named low-var, follows a uniform probability law, combined with the random draw of probability p. It has a low variance which implies that the overall execution time of the function will always remain relatively close to the theoretical mean. The second model, referred as high-var, was specifically designed to have a much higher variance and a comparable mean. It is based on a binomial probability law. The number of inserted instructions, if not null, is 2 to the power of the number obtained from the binomial law. The variables N and p as well as the model can be chosen by the user when choosing a polymorphism configuration. The mean value of a model impacts the overall execution time therefore it should be kept low for the sake of performance, whereas having a high variance increases the attacks complexity [28]. The low-var model is interesting for time constrained applications where the execution time should not vary too much, otherwise the high-var model should be preferred. The high level of configurability of the insertion of noise instructions allows the user to tune the level of variability according to his will.

Table 1. Probability models that control the number of noise instructions to be inserted in between two original instructions. Both models are configurable by the user. The high-var model provides high variance while keeping the mean quite low.

low-var model
$P[X=0]=1-p$
$\forall i \in [1,N], P[X=i]=\frac{p}{N}$

high-var model
$P[X=0]=1-p$
$\forall i \in [0,N[, P[X=2^i]=p\times 2^{-(i+1)}$
$P[X=2^N]=p\times 2^{-N}$

At every insertion of one noise instruction, the SGPC randomly chooses one instruction among add, sub, eor and load instructions. Then, it randomly chooses the operands and allocates a free register for the destination register.

Dynamic noise. We call *dynamic noise* a dynamic mechanism that provides a variable execution from one execution to another even without runtime code re-generation. We use a sequence of noise instructions whose starting point varies from one execution to another thanks to a random branching mechanism: a forward jump whose size is randomly chosen precedes the noise instructions sequence and skips a random number of noise instructions of the sequence. Such sequences for dynamic noises are inserted by the SGPC during runtime code generation following the same procedure as the insertion of noise instructions. The difference resides in the fact that at every insertion of one noise instruction, the SGPC randomly chooses to insert either an add, a sub, an eor, a load, or a dynamic noise sequence instead. Each time the SGPC is executed, different sequences of dynamic noise are generated, at variable locations in the code of the generated polymorphic instance.

This transformation has two advantages. First, it enables to partly decorrelate the executed code to what happens during the generation, preventing an attacker to gain precise knowledge

of the generated code during code generation. Second, as the execution of the same polymorphic instance varies, it enable to lowers the constraints on the frequency of regeneration owing to security requirements. As a consequence, the approach can be used even on systems that cannot afford to regenerate the code too frequently.

The jump size of every jump associated with a dynamic noise sequence of a polymorphic instance must be efficiently determined and must randomly vary from one execution of the polymorphic instance to another one. The size of all jumps associated with dynamic noise sequences of one polymorphic instance should also be different during the same execution. To this end, a register is reserved to hold a random number used to efficiently compute a random jump size at every execution of every dynamic noise sequence throughout an execution of a polymorphic instance. Also, the number of instructions in a dynamic noise sequence can only be a power of two. The jump size is then computed by masking the random value held in the reserved register with an immediate, by using a Boolean and. Figure 2 shows an example of such sequence with 4 noise instructions. The part before the bx handles the branch offset[1], and 4 randomly chosen noise instructions are generated after the bx. Although the number of noise instructions in the sequence is fixed, it is configurable by the user, even within a function. As an example, in our experiments, we selected a dynamic noise sequence size of 32 instructions at the beginning and at the end of the function, and of 4 instructions in the middle. The rationale behind this choice is the need of a higher variability at the beginning and the end of the function making the synchronisation with the function execution more difficult.

```
@ R7 = number of bytes to be skipped
and R7, Rrand, 3 ≪ 2
@ 2 must be added to R7 since, during the
@ add R7 PC R7 instruction, PC points
@ to the bx inst whose size is 2 bytes.
@ Also, the least significant bit must be
@ set in the address (Thumb mode)
@ which gives an offset of 3
add R7, R7, 3
add R7, PC, R7
@ R7 = address of targeted noise inst
bx R7 @ jump into the sequence
add R7, R8, #41 @ 4 noise instructions
xor R8, R10, R7
load R7, R4, #34
add R8, R8, #101
```

Fig. 2. Exemple of sequence of instructions produced with dynamic noise. The value of Rrand changes regularly.

The register that holds random values is managed as follows. A dedicated place is reserved in memory to hold a seed value that will vary from one execution to another one. At the beginning of every execution of the function (polymorphic instance), the stored value of the seed is loaded to initialise a fast PRNG. The fast PRNG is then used to set a new random value in the reserved register. Then, throughout the execution, the register value can be updated by some other noise instructions (e.g, a noise addition instruction inserted by the SGPC can add a random immediate to the register). This makes the value of the register change throughout the execution of the function. Finally, at the end of any execution of the function, the value of the register is stored, it will be used as a seed for the fast PRNG at the beginning of the next execution.

Theoretical number of variants. To give an idea of the variability achievable by our approach, one can easily compute an underestimate of the theoretical number of variants N_v. Considering only the insertion of noise instruction, with the low-var model, $N_v \geq (\sum_{i=0}^{N} 4^i)^{number\_instructions-1}$. The 4 comes from the fact that noise instructions are selected among 4 different instructions (add, sub, eor, load). Taking p=1/7 and N=4 this gives $N_v \geq 341^{number\_instructions-1} > 6\times10^{22}$ for a code of only 10 instructions, and roughly 10^{704} for a code of 278 instructions as the aes T-table used later in the experimental evaluation. This number is an underestimated number of variants as it does only take into account the classic noise (not the dynamic noise nor the other transformations), and it does

[1]For platforms that do not support add with PC, 2 additional instructions are required to compute the branch address.

not take into account the randomness used within the noise instructions (as their immediate values are randomly chosen).

Management of register availability. The insertion of noise instructions and the use of semantic variants may require free registers. A liveness analysis performed statically by the compiler is used to allocate registers at low cost at runtime for these instructions.

During the compilation of SGPCs, Odo performs a static register allocation ignoring any polymorphism aspect. Then, Odo performs a backward register liveness analysis right before code emission of the SGPC. During the code emission of the SGPC, Odo emits additional calls throughout the SGPC's code in order to transfer the liveness information. These calls indicate which registers are free (or not) in between two useful instructions. In our SGPC example in Listing 4, `add` uses `r1`, thus `r1` is alive right before this instruction. Then `sdiv` defines `r1` without using it. Thus `r1` is dead before the `sdiv` instruction and alive right after. As a consequence, `r1` is free to be used (written) between the `add` instruction and `sdiv` one.

Thanks to these calls, the SGPC is made aware of the liveness state at any point of the program. It can then select free registers when needed. All registers allocated for the noise instructions and for the semantic variants are chosen randomly among free registers. As the extra registers used in the sequence of instructions resulting from these polymorphic transformations are dead immediately after their use, the static liveness analysis remains unmodified whatever the extra registers used.

Branch management. Target offsets for branch instructions are computed at runtime, as the insertion of noise instructions and the use of semantic variants make the size of the code vary. If the offset becomes too large for the initially selected encoding, the SGPC chooses another encoding that allows larger target offsets.

5 MEMORY MANAGEMENT

This section presents how program memory is managed in order to make our approach usable in practice on embedded systems with limited memory resources. More precisely, the memory management must offer a solution to the following constraints:

- targeted platforms can be constrained embedded systems without dynamic memory management (i.e., no `malloc`) and that have a limited amount of memory,
- the use of noise instructions and semantic variants makes the size of the generated code vary from one generation to another,
- the instance buffer has to be writable during runtime code generation and executable during execution, but both permissions must not be activated at the same time.

The first constraint makes the answer to the second one not straightforward: the absence of dynamic memory allocation prevents from allocating an instance buffer of the right size at each runtime code generation. Moreover, it is not acceptable to systematically allocate an instance buffer of the possible largest code size, as it would be a huge waste of memory; it would also make the approach unusable on systems with severe memory limitations. Statically allocating an instance buffer whose size is lower than the worst case size implies that buffer overflows could occur at runtime, which threatens both the functionality and the security of the entire platform.

As a solution, we first exploit the static knowledge of the reference assembly instructions of the function in order to:

(1) Limit the amount of memory by statically allocating a realistic amount of memory for instance buffers without impacting the variability obtained with code polymorphism.
(2) Prevent buffer overflows by dynamically guaranteeing that the generated code fits in its instance buffer. Our approach is based on a detection mechanism that adapts the insertion of

noise instructions to the remaining space and to the size of the remaining useful instructions to generate, in case a buffer overflow should occur. Furthermore, we show how to guarantee that the probability of reaching the conditions of a buffer overflow remains below a configurable threshold.

(3) Guarantee that only the legitimate SGPC can write into the instance buffer, and no other parts of the program nor other programs. This is achieved by a dedicated management of the memory permissions of each instance buffer, and leveraged by the specialisation of the SGPCs along with the static allocation of their associated instance buffers.

It is important to allocate a code buffer of realistic size, so that the prevention of buffer overflows does not introduce a bias in the probabilistic models used to implement polymorphism, i.e., so that no vulnerability is introduced. For example, if considering only the insertion of noise instructions: if the allocated size allows not more than the original instructions to be emitted, noise instructions will never be emitted, and the polymorphic instance will not present any behavioural variability. More generally, our objective is to control the likelihood that a bias is introduced in the probability models used in the polymorphic transformations due to the restrictions on the size of the allocated code buffer, in order to avoid exploitations of such biases by attackers.

5.1 Allocation of instance buffers

The allocated size of an instance buffer is computed during the generation of the associated SGPC, by computing the size required for useful instructions and the size required for noise instructions. For useful instructions, Odo computes the sum S_u of the size of the useful instructions, considering the largest semantic equivalents in case of available semantic variant. For noise instructions, Odo computes the size S_n to allocate by considering the probability law SP that results of the $n_i - 1$ draws of law P (the law used to determine the number of noise instructions to be inserted), where n_i is the number of original instructions. Given SP, we can compute the size to allocate so that the probability of having an overflow is below a given threshold: this size S_n corresponds to the size of a noise instruction multiplied by the smallest integer i that checks the condition $\sum_{j=i+1}^{\infty} SP[j] < threshold$.

Odo automatically computes SP from the knowledge of P, and then finds the appropriate size to allocate considering either a threshold provided by the user, or a default threshold set to 10^{-6}. As a result, with the default threshold, the probability of directly generating a code that fits into the allocated memory is more than 999,999 chances over a million, whatever the original size of the code. Thus, the probability to introduce an exploitable bias in the probability models used for the polymorphic runtime code transformations is controlled such that this could not lead to an exploitable vulnerability. Moreover, the SGPC prevents buffer overflows at runtime, as explained in next section, to guarantee that the code always fits into the allocated instance buffer.

The gap in terms of size that results from the proposed allocation policy instead of a worst case allocation policy is huge. Figure 3 illustrates that the gap is asymptotically constant as the number of original instructions increases; for the `high-var` model considered here, and considering one probability draw following P in between each pair of consecutive instructions, the difference of the allocated sizes represents about 58 bytes for each original instruction of the function, while about 10 bytes are saved for each original instruction for the `low-var` model. Considering an original function of 200 instructions, this results in a difference of 2kB for the low-var model and 11.6kB for the high-var model considered.

5.2 Prevention of buffer overflows

S_u, the maximal size of the useful instructions, is statically computed by Odo. At runtime, the SGPC initialises with S_u a variable in charge of keeping track of the remaining needed space for the useful

instructions and the longest semantic variants. This variable is decremented throughout the code generation, after the emission of each useful instruction. Moreover, at every generation of noise instructions, the noise instructions generator computes the maximum number of instructions it can insert by considering this variable and the available buffer space. This information enables the runtime to constrain the generation of noise instructions to guarantee that no overflow can occur.

5.3 Management of the memory access permissions on code buffers

Runtime code generation requires that code buffers are accessed with write and execute permissions, possibly exclusively. However, in embedded systems, write permissions are systematically disabled on program memory to prevent the exploitation of buffer overflows attacks. To overcome this issue, JITs typically provide an access to program memory exclusively with write or execute permissions [14, 15, 25]. In this work, we follow the same approach, but in addition the buffer of the polymorphic instance is protected so that only the legitimate SGPC can write into it.

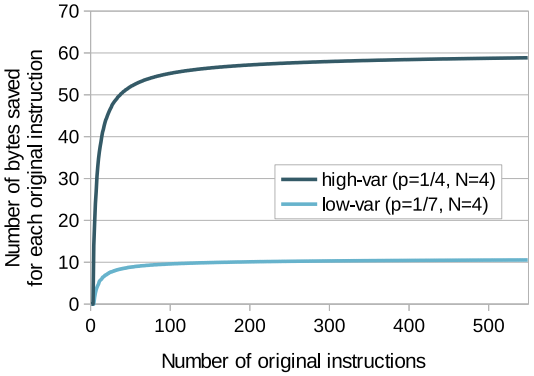

Fig. 3. Difference in terms of bytes per instruction between our policy (with threshold = 10^{-6}) and worst case policy. For the high-var model taken as an example, this difference is approximately equal to 58 bytes per instruction. For a code of 500 instructions, this represents a difference of 29kB.

In this section, we propose a mitigation technique based on the facts that the instance buffers are statically allocated, and that each instance buffer has a unique associated SGPC. We rely on the memory protection unit (MPU) of the target platform to switch access permissions related to the instance buffer from execute-only to write-only (and vice-versa) whenever needed. By default, a code buffer allocated for a function only has the execute permission. At the beginning of the SGPC execution, the SGPC raises an interruption in order to be granted by the write permission (Figure 4). The interruption handler checks the address where the interrupt has been raised. If the check passes, it exchanges the execute permission with the write one only for the instance buffer associated with the requesting SGPC. Thanks to the static allocation of the instance buffers, the interrupt handler is made aware of which memory zone is associated with which SGPC (to allow a switch of permissions only for correct pairs of interruption address and memory zone) and of the addresses of each instance buffer (to switch the permissions for only a buffer zone). At the end of the SGPC, the write permission is removed and the execution permission added by following the same principle. This solution is lightweight (see section 6.3.2), and can be easily extended to systems that provide a MMU instead of a MPU. It is suitable only if the system does not have preemptive multitasking however.

We discuss now the cases where the system support is different from the one we used. For systems with OS that have preemptive multitasking, the OS must be in charge of managing the access permissions to guarantee exclusive access to the code buffers. Just as the interrupt handler in the presented approach, the OS will take advantage of the information statically available to perform legitimacy validation. For any platform without Memory Protection Unit (MPU) nor OS, control flow integrity techniques can be used to ensure that an instance buffer can (1) only be modified by the dedicated SGPC and (2) only be jumped to from the address where the polymorphic code is called,

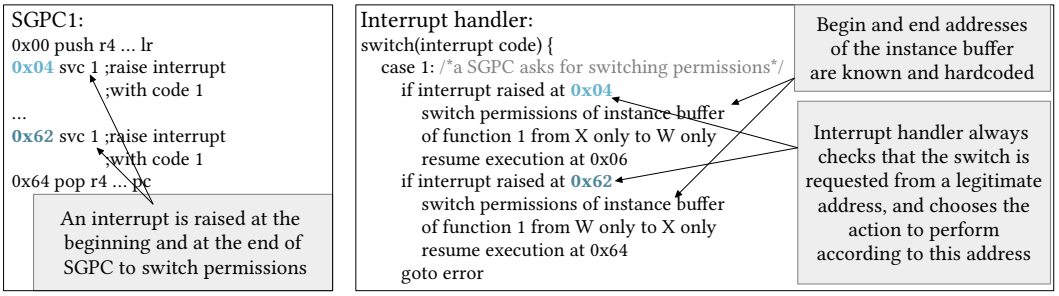

Fig. 4. Permissions handling using interruptions and the MPU (Memory Protection Unit). The instance buffers are never writable (W) and executable (X) simultaneously. Their permissions can be switched thanks to the static allocation of the instance buffers and the specialisation of SGPCs that allow comparing the address where the interrupt has been raised to hardcoded values.

e.g., in the case of Listing 4 the address corresponding to the call `code_f(a,b)`. The principle is to insert checks before each stores and each branches (direct or indirect) in order to verify the validity of any write to a code buffer and of any jump into a code buffer. This idea was presented in the CFI extension called SMAC presented by Abadi et al. [3], and induces a much higher performance cost.

6 EXPERIMENTAL EVALUATION

6.1 Experimental setup

We considered a constrained embedded platform. We used a STM32VLDISCOVERY board from STMicroelectronics, fitted with a Cortex-M3 core running at 24 MHz, 8 kB of RAM, and 128 kB of flash memory. It does not provide any hardware security mechanisms against side-channel attacks.

Our setup for the measurements of electromagnetic emission includes a PicoScope 2208A, an EM probe RF-U 5-2 from Langer and a PA 303 preamplifier from Langer. The PicoScope features a 200 MHz bandwidth and a vertical resolution of 8 bits. The sampling rate is 500 Msample/s (which gives 20.83 samples per CPU cycle), and 24500 samples were recorded for each measurement.

For the study on AES below, both for the t-test and CPA, a trigger signal was set via a GPIO pin on the device at the beginning of the AES encryption, and after runtime code generation by the SGPC, to ease the temporal alignment of the measurement traces. We verified that our measurements covered the full time window of interest, both for the reference and for the polymorphic implementations of AES. Note that the SGPC does not manipulate the secret encryption key, and hence would not be vulnerable to the side-channel attacks used here. Note also that our trigger setup makes the attack easier than it would be in practice for an attacker, as she would have to align the measurements.

Odo is based on LLVM 3.8.0. All C files produced by Odo were compiled using the `-O2` optimisation level, which offers a good compromise between code size and performance. Because of the limited memory available on our target, we had to compile the rabbit and salsa20 benchmarks using the `-O1` optimisation level to produce code whose size is more suitable for the platform. All the programs executed by the platform (the initial ones or the ones generated by Odo) were cross-compiled with the LLVM/`clang` toolchain in version 3.8.0 using the compilation options `-O2 -static -mthumb -mcpu=cortex-m3`. Execution times were measured as a number of processor clock cycles, . The size of the programs (`data`, `text`, and `bss` sections) was measured in bytes with the `arm-none-eabi-size` tool from the gcc toolchain.

As previously explained, the level of variability provided by Odo can be configured with transformations to use and their potential parameters. We first analyse the performance and security

Table 2. Test-cases considered for the evaluation. Our approach is fast and easy to deploy, whatever the nature of the program to be hardened, as it is carried out through compilation.

Nature of benchmark	List of benchmarks
8 block ciphers	AES 8 bits, AES T-table, camellia, 3 des, xtea, present, misty 1, simon.
4 stream ciphers	arc4, rabbit, salsa20, trivium.
2 hash functions	sha256, md5.
1 general function	bytecompare from verifyPin.

of 17 different configurations on the AES T-table, and we discuss the possible trade-offs between security and induced overheads. Then, we apply 4 polymorphic configurations for hardening 15 representative benchmarks (Table 2). These 4 selected configurations include a configuration with polymorphism disabled, and 3 configurations using different variability options. They are used to evaluate the performance and code size overheads for none to high variability at runtime.

In the following, configurations are designated with acronyms of the transformations used, presented in Table 3.

6.2 Use case study: AES

This section presents a performance and security study of the AES implementation from the mbed TLS library [2] (AES T-table in our benchmarks). Nowadays, this implementation is widely used in many embedded systems from IoT devices to mobile and desktop computers, and its original implementation does not feature any countermeasure against side-channel attacks. The security against side-channel attacks of the AES cipher has been studied for several years. It is often used as a reference for security evaluation.

Table 3. Acronyms used for configuration names

RS	register shuffling
IS	instruction shuffling
SV	semantic variants
N1	noise instructions without dynamic noise, probability model `low-var`, p=1/7, N=4
N2	noise instructions without dynamic noise, probability model `high-var`, p=1/4, N=4
DN1	noise instructions with dynamic noise, probability model `low-var`, p=1/7, N=4
DN2	noise instructions with dynamic noise, probability model `high-var`, p=1/4, N=4

6.2.1 *T-test based security evaluation.*

Presentation of the t-test. The t-test is a statistical method applied on two sets of side-channel measurements in order to determine if the two sets are statistically distinguishable. In the case of side-channel attacks, the evidence of distinguishability reveals the presence of an information leakage, which could potentially lead to a successful attack, for example by means of a CPA. The t-test tries to differentiate the means and standard deviation of the two sets of measurements by computing values called *t-statistic*. These values are computed all along the measurement traces. If they remain in between]-4.5,4.5[, the two sets are considered to be statistically non-distinguishable with a confidence of 99.999% [23, 34].

In this paper we use the *non-specific t-test* [34], because it is independent of the underlying architecture and of the model of an attack. The t-test was performed by measuring the electromagnetic emission of our device during the execution of two randomly interleaved groups of 10^4 encryptions. The two groups of measurement traces gather the electromagnetic emissions retrieved during encryptions of random plaintexts and fixed plaintexts respectively.

Analysis of the effects of the transformations on the t-value. We performed 10 times the t-test, for 17 different configurations to evaluate the resulting security of the AES T-table on our platform. For all configurations, the period of regeneration was set to 1. For each process, we picked the maximal t-value (in absolute). In the following, t-value refers to the maximal for one t-test. When this value is bellow 4.5 the configuration passes the t-test.

Figure 5 shows a violin plot of the t-values obtained for the 10 t-tests performed for each configuration. A violin plot shows the minimal, maximal as well as the median of the maximal t-values obtained during the 10 t-tests as well as the distribution of the maximal t-values. Without any protection, the t-test fails and the maximal t-values are very high: the configuration none exhibit a median of the maximum t-values of 110. The 5 polymorphic transformations we deploy have two goals: to introduce desynchronisation (instruction shuffling, semantic variants, noise and dynamic noise) and to change the profile of the leakage (register shuffling, semantic variants, noise and dynamic noise). As more transformations get activated, the number of passed t-tests increases, and the maximal t-values strongly decrease. As polymorphism does not remove the leakage, one could expect all the t-tests to fail. However, we see that as the variability of the code grows, the maximal t-values decrease progressively, and that they quickly reach a point where more t-tests pass than fail. One of the configuration even passes all the 10 t-tests, which means that all t-tests failed to detect the hidden leakage. Note that we limited the number of configurations for the sake of clarity, but as our approach is fully configurable, one could get other configurations that are close to the studied ones, or with even more variability.

Each of the polymorphic transformations used in isolation has a different impact on security (Fig. 5.) Register shuffling seems to have very little impact on the t-value on our platform but may be of interest for other platforms as the register index can have an impact on measurements [35]. Instruction shuffling has an higher impact on reducing the t-value than register shuffling, but has a lower impact than semantic variants. Noise instructions have a higher impact than semantic variants on reducing the t-value, and dynamic noise has the highest impact. When transformations are combined, the resulting effect on the t-value is harder to analyse, their combination can exhibit some complex interactions. For instance, dynamic noise could lower the effect of instruction shuffling as it disables instruction shuffling around the dynamic noise sequences. Yet, alone or combined, dynamic noise seems to be the one that has the higher impact on the t-value: it clearly increases the number of passed t-tests compared to noise instructions, as DN1 and DN2 pass more t-tests than N1 and N2 respectively. We also note that using a stronger noise model (`high-var` instead of `low-var`) improves the observed security; DN2 and N2 show a better security than DN1 and N1 respectively.

The best configuration is the one where all transformations are activated. However, as the impact of transformations depends on the targeted application (mix of instructions, available semantic variants, etc.), the impact of a configuration on security depends on the platform and also on the application.

Analysis of the effects of dynamic noise on the t-value as the period of regeneration increases. We study more precisely the effect of dynamic noise on the observed maximal t-value. In particular, we show that dynamic noise enables the user to increase the period of regeneration way more than the user could using non-dynamic noise.

Figure 6 shows how the median of the maximum t-values gathered on 10 t-tests evolves as ω grows, for the N1, N2, DN1 and DN2 configurations. Please note the log scales, both for y-axis and x-axis. Configurations DN1 and DN2 show better security than N1 and N2 respectively for all the tested period of regenerations. In addition, the configurations where dynamic noise is activated maintain the security well better than the ones where it is not activated. For example, while N2 and DN1 show similar values for small periods of regenerations, the security provided by N2 starts to drop

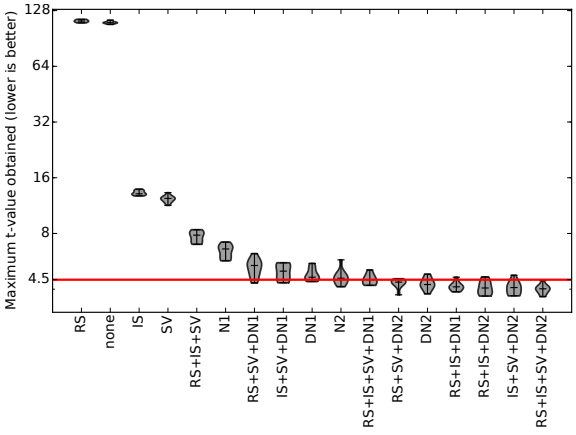

Fig. 5. Maximum t-values obtained for 10 t-tests for 17 different configurations. The distribution of these t-values is represented, as well as the minimum, maximum and median. The configurations are sorted by the medians obtained. Several configurations passed the t-test more times than they failed it, and one configuration passed all the t-tests.

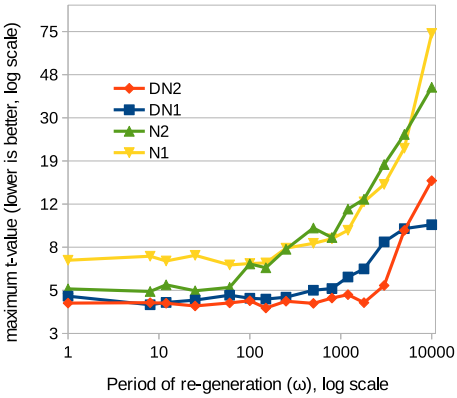

Fig. 6. Observed maximum t-values (taken as a median of the maximum t-values of 10 t-tests) for configurations without and with dynamic noise, in function of the period of regeneration. The configurations with dynamic noise show overall better security, and better preserve security as the period regeneration increases.

from a period of 100, while the security provided by DN1 starts to drop from a period of 1000. The DN2 configuration exhibits an even greater ability to maintain the security level when the period increases, as the t-values observed start to increase significantly only for periods greater than 8000.

Thus, the dynamic noise makes the choice of the period of regeneration wider, which may be useful to lower the generation cost.

6.2.2 Performance and code size overheads. We discuss in this section the impact of the different transformations on performance and code size. We measured performance overheads in terms of relative execution time w.r.t. the reference code with and without taking into account the generation cost. *Execution time overhead* denotes the ratio of the averaged execution time of 10^4 variants without the code generation cost to the execution time of the reference code. *Global overhead* refers to the ratio of the execution time averaged with 10^4 variants, generation time included, to the reference execution time. For evaluating the global overheads, we considered different regeneration periods. We also measured the code size overhead as the relative code size w.r.t. the reference code.

Execution speed and code size. The Table 4 presents the execution time overheads, global overheads with various period of regeneration, generation time in clock cycles and size overheads obtained for the 17 configurations.

The global overheads are prohibitive when $\omega = 1$, this period of regeneration is probably of interest only if the generation cost can be hidden during waiting times. Yet, the global overheads become more reasonable as the period of regeneration is increased.

The different transformations influence differently the overheads. Activating instructions shuffling in addition to some other transformations roughly doubles the generation time. For instance, the generation time is 82077 cycles for RS+SV+DN1 163873 cycles for RS+IS+SV+DN1, and 107820 cycles for RS+SV+DN2 218909 cycles for RS+IS+SV+DN2. Yet, this transformation has very little influence on execution time overhead. It also has little influence on size overhead. Thus, it is of interest if generation cost can be hidden, or if the period of regeneration is large enough to minimize its impact

Table 4. Overheads obtained in execution time and size for all configurations. "Execution" corresponds to the execution time overhead, "Global" to the global overhead, "Gen time" to the generation time in clock cycles, and "Size" to the size overhead. ω corresponds to the period of regeneration.

Configuration	RS	none	IS	SV	RS+IS +SV	N1	RS+SV +DN1	IS+SV +DN1	DN1
Execution	1.48	1.34	1.34	2.09	2.53	2.08	3.56	3.07	2.31
Global ω=1	8.39	6.91	37.51	17.04	70.41	43.38	67.43	115.44	49.58
Global ω=25	1.76	1.57	2.79	2.68	5.24	3.73	6.12	7.56	4.20
Global ω=100	1.55	1.40	1.70	2.24	3.20	2.49	4.20	4.19	2.78
Global ω=250	1.51	1.37	1.49	2.15	2.80	2.24	3.82	3.52	2.50
Global ω=1000	1.49	1.35	1.38	2.10	2.59	2.12	3.62	3.18	2.35
Gen time (cycles)	8,876	7,147	46,476	19,216	87,228	53,077	82,077	144,402	60,750
Size	1.54	1.38	1.62	1.56	1.84	2.11	2.41	2.36	2.11

Configuration	N2	RS+IS +SV+DN1	RS+SV +DN2	DN2	RS+IS +DN1	RS+IS +DN2	IS+SV +DN2	RS+IS +SV+DN2
Execution	3.39	3.60	4.97	3.71	2.49	3.90	4.48	5.02
Global ω=1	58.93	131.13	88.87	71.06	99.29	143.25	159.57	175.38
Global ω=25	5.61	8.71	8.32	6.41	6.36	9.48	10.68	11.83
Global ω=100	3.95	4.88	5.81	4.39	3.46	5.30	6.03	6.72
Global ω=250	3.61	4.11	5.30	3.98	2.88	4.46	5.10	5.70
Global ω=1000	3.45	3.73	5.05	3.78	2.59	4.04	4.64	5.19
Gen time (cycles)	71,366	163,873	107,820	86,540	124,391	179,065	199,283	218,909
Size	2.19	2.43	2.49	2.18	2.34	2.42	2.44	2.51

on performance. Activating register shuffling has a low impact on both execution time and the lowest impact on the global overhead as it benefits from the static analysis performed during the generation of the SGPC.

Semantic variants impact both the execution time, generation time and size overhead. Its impact has to be considered specifically for a use-case, as from one use-case to another the number of instructions (and their positions in the code) for which variants are available varies. Finally, the overheads due to noise and dynamic noise depend a lot on the probability law P used. For instance, the frequency at which the generator executes the code responsible for inserting strictly more than 0 noise instructions directly depends on the value of p.

Impact of dynamic noise on overheads. Figure 7 shows the global overheads obtained with and without dynamic noise when ω varies from 1 (top left) to 10000 (bottom right) plotted as a function of the security provided (maximal t-value). The sweet spot is at the bottom left, where hardened codes are the most secure and have the lowest overheads.

The configurations with dynamic noise get closer to the sweet sport than the ones with classic noise. In particular, at a given global overhead, they offer a much better security than the configuration with classic noise. This shows that dynamic noise relaxes the security constraints on the period of regeneration. Thus, dynamic noise allows to reduce the regeneration period i.e. the code generation cost.

6.2.3 Trade-off between security and performance. The configurability of our approach allows to explore different possibilities to find a trade-off that fits the user's constraints, by measuring performance overheads and security levels (with a metric like the maximum t-values for instance) for one's particular platform and application.

To find a good trade-off, the user can start by selecting some configurations considering the code generation impact on his application and its performance constraints. For instance, if the generation cost can be hidden, then instructions shuffling really is an interesting option as it increases security without increasing execution time. On the other hand, if the generation cost cannot be hidden, this option may be eliminated to limit the global overhead, and dynamic noise is a better choice as it enables to choose a larger re-generation period.

Then, the user can make some measurements of performance and security on his platform for the selected configurations. Then he can eliminate the configurations that do not match his constraints in terms of performance (e.g. the ones that lead to a too slow execution or a too large memory overhead). Finally, he can choose the configuration that shows the best security and can evaluate it with other security metrics (a CPA success rate for instance) if he wants.

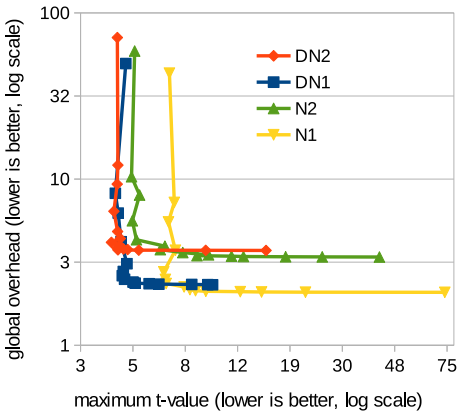

Fig. 7. Global overheads obtained with increasing ω values with and without dynamic noise in function of the obtained maximal t-value. The configurations with dynamic noise show similar or better overheads at a given security level compared to the ones with classic noise. Dynamic noise allows to reach much better security at a given overhead.

For the rest of the evaluation conducted in this paper, we chose 4 configurations that have quite different characteristics. First, we chose the configuration none (no variability) as it allows to show the minimal overheads induced by the generation and the execution of runtime generated code. Then, we considered a configuration named `low` suitable for a highly constraint environment. The `low` configuration includes DN1 with *omega* = 250, as it has limited performance impact and is on the bottom left of the DN1 curve of Figure 7. Then, we chose the configuration RS+IS+DN1 as a `medium` configuration, that supposes that the generation cost is hidden during waiting time. It has an execution time overhead close to the one of the `low` configuration, but showed a better security level. It induces however a generation time way larger than the `low` configuration. Finally, we chose the RS+IS+SV+DN2 configuration as a `high` configuration, because this configuration passed all the t-tests. Its impact on execution time and on generation time is much larger than the other configurations.

The parameters of the selected configurations are recalled hereafter:

none - No variability option.

low - Activated options are register shuffling, insertion of noise instructions with the probability model `low-var` (p=1/7, N=4) with dynamic noise. The regeneration period ω is set to 250.

medium - All mechanisms are activated except semantic variants. The insertion of noise instructions uses the probability model `low-var` (p=1/7, N=4) with dynamic noise and the regeneration period ω is set to 1.

high - All mechanisms are activated. The insertion of noise instructions uses the probability model `high-var` (p=1/4, N=4) with dynamic noise and the regeneration period ω is set to 1.

6.2.4 CPA based security evaluation.

Methodology. We performed a first order CPA against both the reference implementation and an implementation protected using the `low` configuration. The considered attack targeted the output of the first SubBytes function of the AES encryption. The conducted attack aimed at the retrieval of the

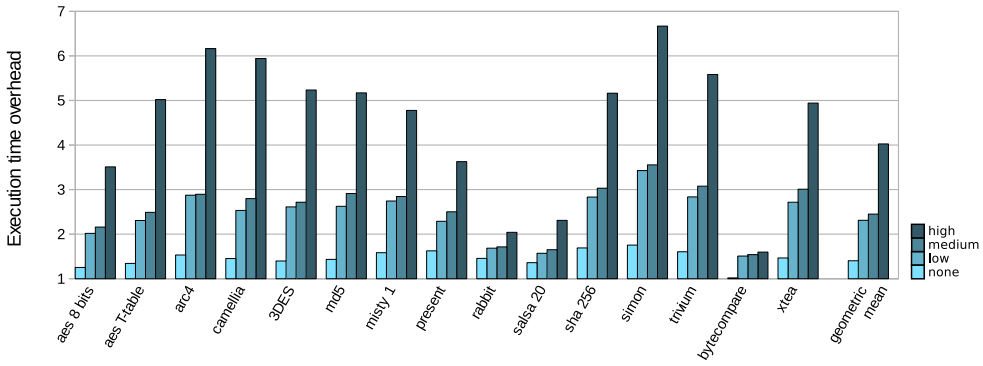

Fig. 9. Execution time overhead compared to statically compiled version (clang 3.8.0 -O2).

first byte of the key as retrieving the other key bytes can be similarly performed with a similar attack complexity. We used the hamming weight as model of the EM emission of the SubBytes.

Results. Figure 8 presents the success rate obtained for CPA against both the reference unprotected AES and an implementation protected by Odo with the configuration low. The success rate represents the statistical proportion of attacks to succeed given a number of traces. A success rate of 1 means that all attacks succeed. The results show that the reference implementation is highly vulnerable, as a success rate of 0.8 (80% of attacks are successful) is reached with about 290 traces on our test platform. Such a number of traces is considered as very low for side-channel attacks, even on unprotected implementations. Furthemore, using 290 traces, the correct key leaks with a correlation value of 0.53, which suggests that our measurement setup provides very good attack conditions.

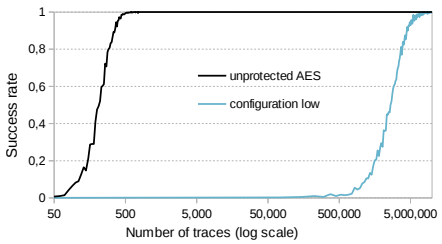

Fig. 8. Success rate of CPA for both the unprotected implementation and the configuration low. The number of traces required for the attack is multiplied by 1.3×10^4, the execution time is multiplied by 2.5 (including generation cost).

The implementation protected by Odo with the configuration low is far more secure, as 3.8×10^6 traces have to be collected to reach a success rate of 0.8, in the same experimental conditions as for the reference unprotected implementation. Compared to the reference, that represents 1.3×10^4 folds more measurements.

6.3 Performance evaluation

The performance evaluation has been conducted by considering 15 different test cases, presented in Table 2. The benchmarks considered for the evaluation are mostly cryptographic benchmarks, either from the mbedtls library [2], the eSTREAM project [1] or a home-made 8-bits implementation of AES. As any C function could be secured by Odo, these selected cryptographic functions only represent a tiny panel of the possible usages of our approach. Hence, we also considered the bytecompare function from a verifyPin implementation from FISSC [20]. We evaluated these test cases against the 4 polymorphic configurations previously selected. We first analyse the execution time overhead (of the polymorphic variants). Then we present an analysis of code generation speed and size overheads.

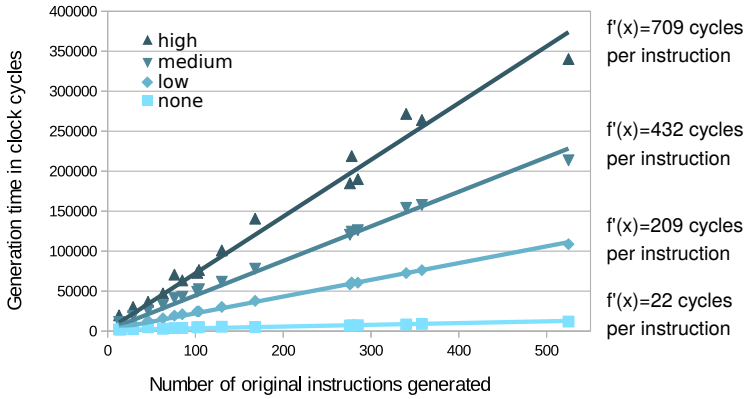

Fig. 10. Generation time vs number of useful instructions generated. The slopes represent the cost of generation per useful instruction. Our approach can easily scale as the functions get bigger.

6.3.1 Execution time overhead. Figure 9 presents the execution time overhead (as defined in Section 6.2.2) for the 15 benchmarks. Depending on the benchmarks and on the configuration, execution time overheads range from 1, which means that the hardened code executes as fast as the reference one, up to 7. Execution time overhead depends on the instruction mix of the initial code. First, as semantic variants are not available for all instructions, it impacts more the performance of some codes than others. Second, as the code is generated in RAM, in our platform, instruction fetch and data loads use the same bus. As a consequence, load instructions take a longer time to execute than if the code was in Flash. More precisely, when stored in RAM a Thumb 1 (16 bits) load instruction takes 1.5 cycles in average and a Thumb 2 (32 bits) load instruction takes 2 cycles (instead of 1 cycle when stored in Flash memory).

The overheads observed in Figure 9 indicate that the code produced remains generally efficient considering the amount of variability that it gets. The overall impact of execution time overhead for an application which has one or several polymorphic functions depends a lot on the proportion of the execution time initially spent in the transformed functions. The different configuration possibilities allow to adapt the polymorphism to the application requirements.

6.3.2 Generation speed. In order to analyze the cost of code generation, we correlated the number of useful instructions to the time needed for code generation for all the considered configurations (Figure 10). We also measured the minimal cost of code generation which can be obtained with a call to an SGPC with no instruction to generate. This minimal cost is about 1500 cycles. This cost includes the time taken for changing the instance buffers permissions with the MPU ($\approx$100 cycles per generation). Figure 10 shows that the generation time evolves almost linearly with the number of useful instructions to generate. The reader should note that the number of useful instructions only depends on the source program and the optimisation level, but does not depend on the polymorphism options. The slopes of the trend lines in Figure 10 indicate the mean number of cycles required to generate one useful instruction. For the configuration `high`, the generation of each instruction requires about 709 cycles, and only 22 cycles per instruction when no polymorphism is introduced. As an indication, static compilers like LLVM take about 3 million cycles per instruction [13], while the specialized code generator deGoal requires 233 cycles per instruction [12, 13]. Thus, the runtime code generation is really efficient.

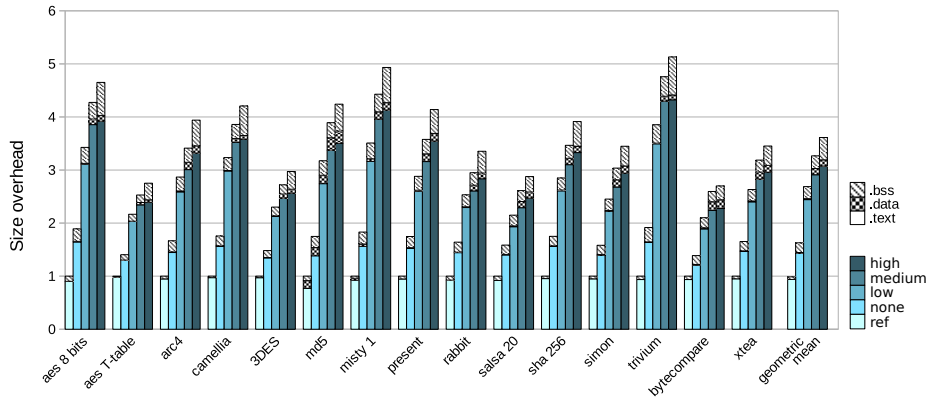

Fig. 11. Code size overhead compared to statically compiled version (clang 3.8.0 -O2).

6.3.3 Size overhead. Figure 11 illustrates the size overhead computed as the ratio between the size of the protected binary and the size of the unprotected binary. Each bargraph gives the breakdown of the `.text`, `.bss`, and `.data` sections. The results show that increasing the variability increases the size overhead. This is due to several factors. First, as new mechanisms are enabled, the library and SGPC codes that handle these mechanisms are added into the binary, which impacts `.text` sections. Second, a greater polymorphic variability generates larger instance buffers, as noise instructions are more probable or as the use of semantic equivalences requires to keep more space for individual instructions, which impacts `.bss` sections. The increase of the `.data` sections is due to the use of a shuffling buffer for instruction shuffling, and to some private data of the Odo-runtime library.

The overhead in code size may appear too high for constrained systems, however we considered only benchmarks that correspond to one functionality of an embedded system. By limiting the part of the embedded software to be secured for a given product to the minimum one, the overhead should be far lower. Also, if several polymorphic functions are deployed on a same platform, the library could also be shared among the SGPCs. Moreover, all benchmark were able to fit in our platform which has 8kB of RAM and 128kB of flash memory.

6.3.4 Conclusion of the performance evaluation. The evaluation shows that our approach is generic. Generation speed evolves linearly with respect to the number of original instructions of the code, and is much higher than compilation speed of static compilers. The execution time overheads obtained can be acceptable. Indeed, first order masking is a widely used countermeasure that induce overheads that can easily range from 20 to 2000 [10]. As masking is used in practice in spite of these overheads, hence the overheads induced by our approach can be considered as acceptable too. The configurability of our approach allows then to tune performance/security trade-off. The security evaluation on the configuration `low` showed that the CPA required 13000 times more traces than on the reference, while the global execution time overhead is only 2.5.

7 DISCUSSION

In this section, we discuss and compare our approach with the closest existing approaches.

The Odo-runtime library is an enhanced version of COGITO, a runtime code generation framework which already provides support for random register allocation, instruction shuffling, semantic equivalences, and insertion of noise instructions [18]. COGITO requires the developer to implement polymorphic components using a low-level domain specific language (DSL). This approach brings

Table 5. Comparison of execution time overheads for Odo and Code Morphing [5] and MEET [7]. Elements in the same line have close security levels. Our approach can be used with much lower overhead (50% saved) than previous automated runtime approach [5], and with much lower overhead (27% saved) than the static approach MEET when the generation cost can be hidden. When generation cost cannot be hidden, overheads with Odo high are smaller than MEET overheads even with a small period of regeneration like $\omega = 100$

Benchmark. ω is the regeneration period	AES T-table	camellia	3DES	misty1	present	xtea	geo. mean
Code Morphing [5] ω=100	**5.00**	-	-	-	-	-	5.00
Odo `low` ω=250	**2.50**	2.75	2.77	2.94	2.29	2.82	2.67
MEET [7]	6.76	8.99	8.61	10.1	2.79	6.02	**6.68**
Odo `high`, no code gen.	5.01	5.94	5.23	4.78	3.63	4.94	**4.87**
Odo `high` ω=100	6.72	7.97	6.71	6.30	3.63	5.70	6.00
Odo `high` ω=1000	5.19	6.14	5.38	4.93	3.63	5.01	4.99

Table 6. Comparison of size overheads for Odo and MEET [7]. Size overheads of the Code Morphing approach [5] are unknown. Size overheads obtained with our approach are much smaller (29% saved) than the ones of the static approach MEET.

Benchmark	AES T-table	camellia	3DES	misty1	present	xtea	geo. mean
MEET [7]	6.19	7.82	9.80	4.49	2.90	3.14	**5.18**
Odo `high`	2.75	4.20	2.97	4.93	4.13	3.45	**3.66**

more flexibility, but requires to re-implement the polymorphic component with the provided DSL. The Odo approach differs from this previous work by offering an automatic compilation of SGPCs from annotated C source code, a precise way to estimate a realistic code size for static memory allocation as well as a dynamic mechanism for preventing overflows during runtime code generation, and a memory protection mechanism using the MPU. Furthermore, our implementation of register shuffling relies on the static register allocation of LLVM/clang, which provides a code of better quality than the runtime register allocation used in [18].

Code Morphing [5] and MEET [7] are close works that proposed an automated approach to deploy polymorphism. Code Morphing relies on dynamic code modification and a compiler to apply code polymorphism. The polymorphic engine randomizes semantic equivalences and register uses, shuffles instructions and performs array access permutations. However, it does not provide any solution for the management of memory permissions. This has been stressed by the authors of MEET as a motivation for another approach that removes the need for both writable and executable code segments. MEET relies on an automatic and static generation of multiple semantic variants for small sequences of instructions of the code to harden. Variants are then randomly selected at execution time. This approach allows to use code polymorphism without the need of runtime code generation but, as a static approach, may suffer from high size overheads. Our approach uses dynamic code generation but provides mitigations for security concerns related to this use.

Tables 5 and 6 give the overall execution time and the size overheads obtained with the different approaches. We give here some comparison using configurations that are close in terms of security. However, the reader should note that the security evaluation performed in each paper (including ours) has been carried out on different platforms, which may present different side-channel profiles, hence presenting different levels of resistance to the CPA used in the related experimental studies. Thus, comparisons of the security level achievable with the different approaches are delicate. To compare Odo with Code Morphing, we chose the `low` configuration that passes the CPA done for

Code Morphing, on our platform that is easier to attack (because the reference AES is broken in less traces on our platform). For comparison with MEET, we chose the `high` configuration as both this configuration and MEET's AES pass the t-test.

Code Morphing induces important overheads when the regeneration period is small. Considering the execution time of an AES T-table as a reference T_{ref} [2], we estimated that the time taken by the Code Morphing transformations (the process equivalent to our code regeneration) is about $393 \times T_{ref}$ while Odo's configuration `low` takes $47 \times T_{ref}$, which is almost an order of magnitude more efficient.

Compared to MEET, our approach provides smaller code size overheads. Yet, MEET does not suffer from the execution time overhead incurred by runtime code generation. Using Odo, the execution time overheads can be smaller if runtime code generation can be partly or totally hidden, e.g., by executing the SGPC concurrently to other computations. In addition, the security levels obtained by MEET are impressive. In their approach, polymorphism is combined with mask refreshing to remove information leakage from memory accesses. In Odo, memory accesses are blurred by introducing noise memory accesses, but the information leakage due to memory accesses is still present in side-channel observations. Still, our approach allows to reduce the amount of information leakage to the point that the remaining information leakage is not detected by a t-test, without using masking techniques. Our approach could as well be combined with masking, for a greater level of security, but also at the expense of greater overheads. Finally, if the execution time of runtime code generation cannot be hidden, and for comparable execution time overheads, the security level provided by the MEET approach is probably greater than our configuration `high` with $\omega = 100$.

8 RELATED WORK

Amarilli et al. first coined the use of polymorphism by software means as a countermeasure against side-channel attacks [8]. Then, the approaches presented in the previous section have been proposed [5, 7, 18]. As we already deeply discussed them we do not discuss them in this section.

Code polymorphism has also been employed outside the domain of power/electromagnetic side-channel attacks. *librando* hardens a JIT implementation [24]; it randomly inserts `nop` instructions and masks constant values with boolean masks (named *constant blinding*) to avoid code injection in code constants. Crane et al. also propose to randomly insert `nop` and load instructions to perturb cache-based side-channel attacks [19]. In our approach, the risk of code injection is void because the SGPC has no bytecode input; plus, we are able to insert noise instructions of the same nature than useful machine instructions in the generated code. Thus, our approach is probably of interest for cache-based side-channel attacks too, and future work will study its effectiveness.

Compilation has also been used to automatically apply other countermeasures against side-channel attacks. Agosta et al. proposed to bring out several key hypotheses instead of one during an attack so that the attacker cannot determine which one is the right hypothesis [6].

Several approaches also leveraged compilation to automatically apply boolean masking. Eldib et al. used an SMT solver in combination with a compilation flow to automatically compute and apply an effective masking scheme [22]. Moss et al. proposed to automatically insert a boolean masking countermeasure at compile time; a DSL with an dedicated type system allows to describe the level of confidentiality of variables [30]. Agosta et al. proposed a data-flow analysis to determine the vulnerability level of symmetric cryptography primitives [4]. The vulnerability level is a complexity metric based on the number of key bits involved in the computation of each intermediate value; then, the compiler applies a masking countermeasure only to the most vulnerable parts of the secured code to reduce the overall overhead. Bayrak et al. proposed an approach using decompilation and

[2] Unfortunately, the data available in the Code Morphing paper only enabled us to compute the execution time taken by morphing actions as a function of the reference execution time, so we cannot give a more precise comparison.

compilation to apply boolean masking to a binary program [11]. The proposed tool applies the countermeasure on machine instructions that were identified as vulnerable in a preliminary static leakage analysis of the original binary program, and applies a random precharging countermeasure. Luo et al. used a compiler and a SAT solver to automatically generate a threshold implementation [27].

Masking and hiding countermeasure can be used in combination to increase the security level. Our compiler-based approach could be combined with compiler-based masking approaches to get a masked polymorphic code.

9 CONCLUSION

In this paper, we presented an automatic approach to secure code against side-channel attacks with code polymorphism, implemented by runtime code generation. From an annotated source code, our automatic hardening approach implemented in the Odo tool based on LLVM generates specialised code generators for each function to harden. Specialisation of code generators enables to lower the countermeasure cost. Our compilation-based approach enables to optimize the code to produce, to make available static information used at runtime by the code transformations as well as to finely manage memory that host the generated code. Several transformations applied at runtime make the code vary between runtime code generations. We also proposed the dynamic noise transformation to introduce variability between two consecutive executions of the same generated code and to reduce the frequency of code generation.

Experiments showed that the security level can be strongly increased compared to unprotected implementations while keeping the overhead low enough. The flexibility offered by our configurable tool enables a user to meet or trade-off its security and performance requirements. The range of polymorphic variability goes from none to a very high level. The size overhead is lowered compared to static multiversionning approaches. Our runtime code generation is very efficient, about one order of magnitude faster compared to the state-of-the-art, which allows our approach to induce performance overheads that are competitive or smaller than static multiversionning approaches.

ACKNOWLEGEMENTS

We thank Olivier Debicki for his fruitful help on the management of memory permissions, Philippe Jaillon for the preliminary discussions on attack paths on polymorphic implementations, and our reviewers for the many suggestions to improve the initial version of our paper. This work was partially funded by the French National Research Agency (ANR) as part of the projects COGITO and PROSECCO, respectively funded by the programs INS-2013 under grant agreement ANR-13-INSE-0006-01 and AAP-2015 under grant agreement ANR-15-CE39.

REFERENCES

[1] eSTREAM: the ecrypt stream cipher project. http://www.ecrypt.eu.org/stream/.
[2] mbedTLS library. https://tls.mbed.org/.
[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM TISSEC 13*, 1 (2009).
[4] Agosta, G., Barenghi, A., Maggi, M., and Pelosi, G. Compiler-based side channel vulnerability analysis and optimized countermeasures application. *DAC* (2013), 1–6.
[5] Agosta, G., Barenghi, A., and Pelosi, G. A code morphing methodology to automate power analysis countermeasures. *DAC* (2012), 77–82.
[6] Agosta, G., Barenghi, A., Pelosi, G., and Scandale, M. Information leakage chaff: feeding red herrings to side channel attackers. *DAC* (2015).
[7] Agosta, G., Barenghi, A., Pelosi, G., and Scandale, M. The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE TCAD 34*, 8 (2015), 1320–1333.
[8] Amarilli, A., Müller, S., Naccache, D., Page, D., Rauzy, P., and Tunstall, M. Can code polymorphism limit information leakage? *LNCS 6633* (2011), 1–21.

[9] Avirneni, N. D. P., and Somani, A. K. Countering Power Analysis Attacks UsingReliable and Aggressive Designs. *IEEE TOC 63*, 6 (June 2014), 1408–1420.

[10] Barenghi, A., and Pelosi, G. An enhanced dataflow analysis to automatically tailor side channel attack countermeasures to software block ciphers. *CEUR Workshop Proceedings 1816* (2017), 8–18.

[11] Bayrak, A. G., Regazzoni, F., Novo, D., Brisk, P., Standaert, F.-X., and Ienne, P. Automatic application of power analysis countermeasures. *IEEE TOC 64*, 2 (2015), 329–341.

[12] Charles, H.-P., Couroussé, D., Lomüller, V., Endo, F., and Gauguey, R. deGoal a tool to embed dynamic code generators into applications. *LNCS 8409* (2014), 107–112.

[13] Charles, H.-P., and Lomüller, V. Is dynamic compilation possible for embedded systems? *SCOPES* (2015), 80–83.

[14] Chen, P., Fang, Y., Mao, B., and Xie, L. JITDefender: A defense against JIT spraying attacks. *IFIP AICT 354* (2011), 142–153.

[15] Chen, P., Wu, R., and Mao, B. JITSafe: A framework against just-in-time spraying attacks. *IET Information Security 7*, 4 (2013), 283–292.

[16] Coron, J.-S., and Kizhvatov, I. An Efficient Method for Random Delay Generation in Embedded Software. *CHES 5747* (2009), 156–170.

[17] Coron, J.-S., and Kizhvatov, I. Analysis and Improvement of the Random Delay Countermeasure of CHES 2009. *CHES* (2010), 95–109.

[18] Couroussé, D., Barry, T., Robisson, B., Jaillon, P., Potin, O., and Lanet, J.-L. Runtime code polymorphism as a protection against side channel attacks. *WISTP 9895* (2016), 136–152.

[19] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. *NDSS* (2015), 8–11.

[20] Dureuil, L., Petiot, G., Potet, M.-L., Le, T.-H., Crohen, A., and de, C. FISSC: A fault injection and simulation secure collection. *LNCS 9922* (2016), 3–11.

[21] Durvaux, F., Renauld, M., Standaert, F.-X., van Oldeneel tot Oldenzeel, L., and Veyrat-Charvillon, N. Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models. *CARDIS* (2013), 123–140.

[22] Eldib, H., and Wang, C. Synthesis of Masking Countermeasures Against Side Channel Attacks. *CAV* (2014), 114–130.

[23] Goodwill, G., Jun, B., Josh, J., Pankaj, R., et al. A testing methodology for side-channel resistance validation. *NIST non-invasive attack testing workshop* (2011).

[24] Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Librando: Transparent Code Randomization for Just-in-time Compilers. *CCS-SIGSAC* (2013), 993–1004.

[25] Jauernig, M., Neugschwandtner, M., Platzer, C., and Comparetti, P. Lobotomy: An architecture for JIT spraying mitigation. *ARES* (2014), 50–58.

[26] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. *LNCS 1666* (1999), 388–397.

[27] Luo, P., Athanasiou, K., Zhang, L., Jiang, Z. H., Fei, Y., Ding, A. A., and Wahl, T. Compiler-Assisted Threshold Implementation against Power Analysis Attacks. *ICCD* (Nov. 2017), 541–544.

[28] Mangard, S., Oswald, E., and Popp, T. *Power Analysis attacks: Revealing the secrets of smart cards.* 2007, pp. 1–337.

[29] Moos, T., and Moradi, A. On the easiness of turning higher-order leakages into first-order. *COSADE 10348 LNCS* (2017), 153–170.

[30] Moss, A., Oswald, E., Page, D., and Tunstall, M. Compiler assisted masking. *LNCS 7428* (2012), 58–75.

[31] O'Flynn, C., and Chen, Z. Power Analysis Attacks Against IEEE 802.15.4 Nodes. *COSADE* (2016), 55–70.

[32] Ronen, E., O'Flynn, C., Shamir, A., and Weingarten, A.-O. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. Tech. Rep. 1047, 2016.

[33] Sasdrich, P., Moradi, A., and Güneysu, T. Hiding higher-order side-channel leakage. *CT-RSA* (2017), 131–146.

[34] Schneider, T., and Moradi, A. Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. *WISE*, 207 (2015).

[35] Seuschek, H., and Rass, S. Side-channel leakage models for risc instruction set architectures from empirical data. *EUROMICRO DSD* (Aug 2015), 423–430.

[36] Singh, A., Kar, M., Mathew, S., Rajan, A., De, V., and Mukhopadhyay, S. Exploiting on-chip power management for side-channel security. *DATE* (Mar. 2018), 401–406.

[37] Timmers, N., Spruyt, A., and Witteman, M. Controlling PC on ARM Using Fault Injection. *FDTC* (2016), 25–35.

[38] Yu, W., and Kose, S. Exploiting Voltage Regulators to Enhance Various Power Attack Countermeasures. *IEEE TETC 6*, 2 (Apr. 2018), 244–257.